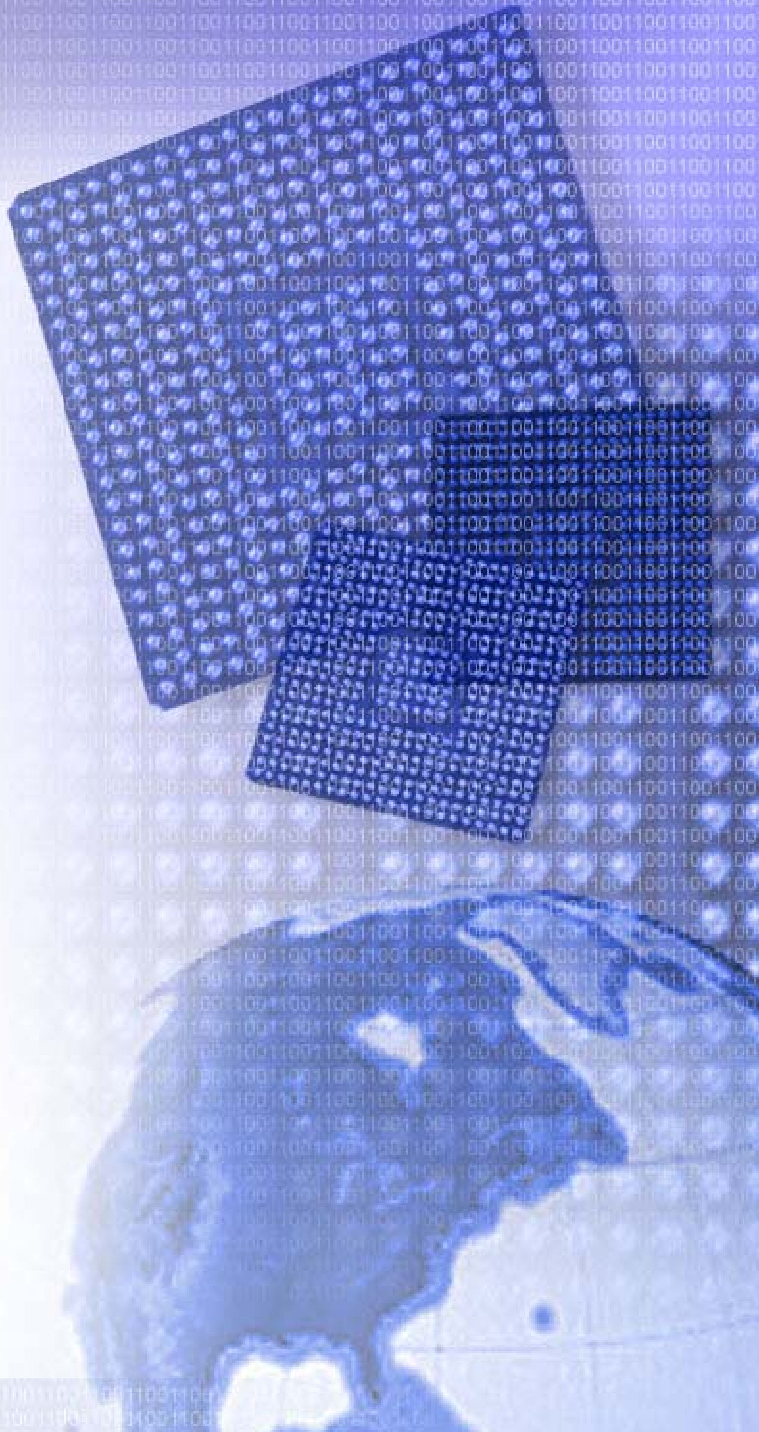
HDL Coding Style



Objective

After completing this module, you will be able to:

- Select a proper coding style to create efficient FPGA designs
- Specify Xilinx resources that need to be instantiated for various FPGA synthesis tools

Outline

- **Introduction**
- Coding Tips
- Instantiating Resources
- Summary
- Appendix:
  - Inferring Logic and Flip-flop Resources
  - Inferring Memory
  - Inferring I/Os and Global Resources

Instantiation Versus Inference

- **Instantiate a component when you must dictate exactly which resource is needed**
  - Synthesis tool is unable to infer the resource
  - Synthesis tool fails to infer the resource
- **Xilinx recommends inference whenever possible**
  - Inference makes your code more portable
  - Synthesis tools cannot estimate timing delays through instantiated components
- **Xilinx recommends the use of the CORE Generator™ System to create ALUs, fast multipliers, FIR Filters, etc. for instantiation**

Outline

- Introduction
- **Coding Tips**
- Instantiating Resources
- Summary
- Appendix:
  - Inferring Logic and Flip-flop Resources
  - Inferring Memory
  - Inferring I/Os and Global Resources

Multiplexers

- **Multiplexers are generated from IF and CASE statements**
  - IF/THEN statements generate priority encoders
  - Use CASE to generate complex encoding
- **There are several issues to consider with a multiplexer**
  - Delay and size
    - Affected by the number of inputs and number of nested clauses to an IF/THEN or CASE statement
  - Unintended latches or clock enables
    - Generated when IF/THEN or CASE statements do not cover all conditions

Avoid Nested IF and CASE Statements

- **Nested IF or CASE statements infer cascaded logic**
  - **More levels of logic → lower performance**
- **When nested IFs are necessary, put critical input signals on the first (outer) IF statement**
  - The critical signal ends up in the last logic stage

Unintended Latch Inference

- **In IF/CASE statements, latches are inferred when:**
  - All possible input values are not covered
  - All outputs are not defined in all branches
- **When the IF/CASE statement is in a clocked process (VHDL) or always block (Verilog), latches are not inferred**
  - Clock enables are inferred instead
  - This is not “wrong,” but it might generate a long clock-enable equation
- **Use default assignments before the IF/CASE statement to prevent latches or inferred clock enables**

Clock Enables

- **Coding style will determine if clock enables are used**
- **Reset and set have precedence over clock enable**

VHDL

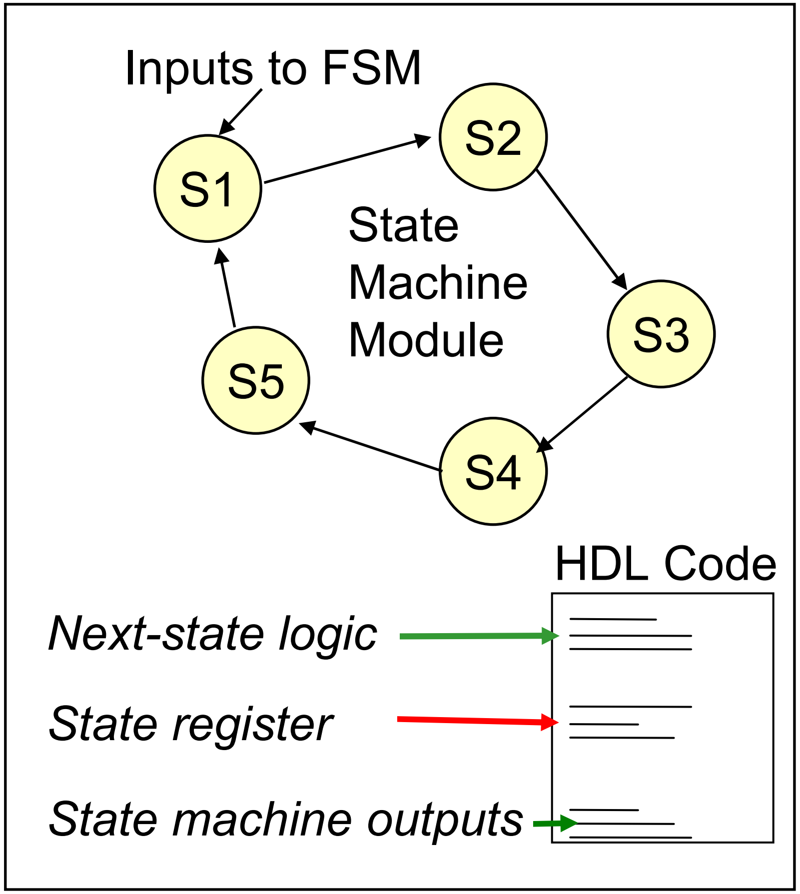
```
FF_AR_CE: process(ENABLE,CLK)
begin
 if (CLK'event and CLK = '1') then
  if (ENABLE = '1') then
   Q <= D_IN;
  end if;
 end if;
end process
```

Verilog

```
always @(posedge CLOCK)
 if (ENABLE)
  Q = D_IN;
```

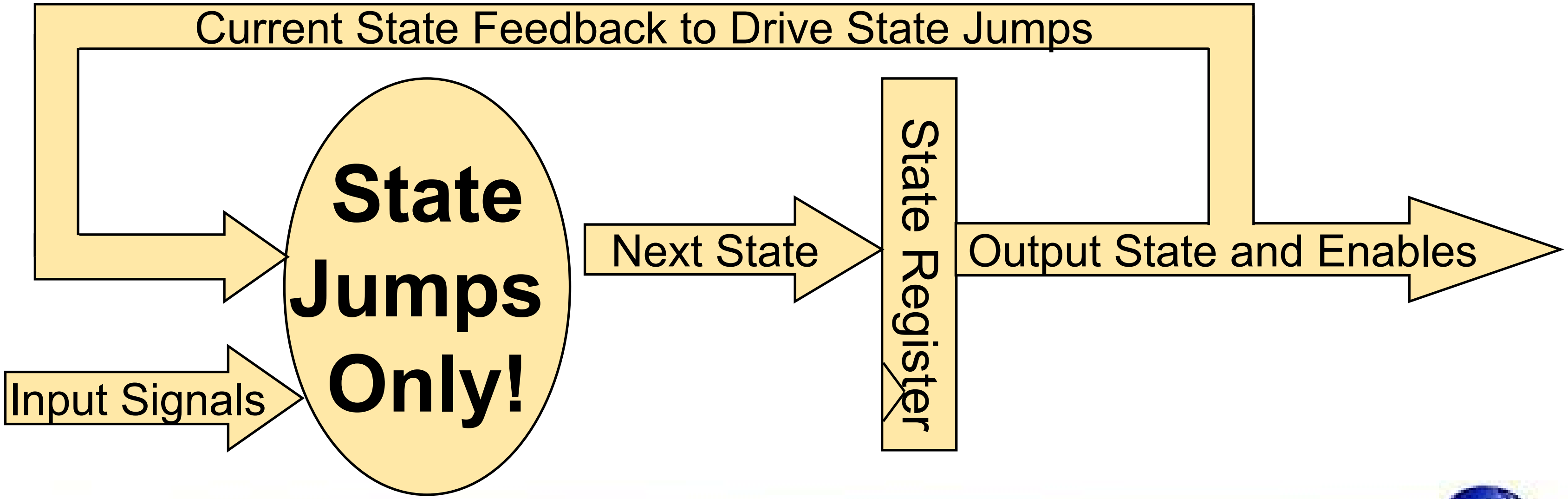
State Machine Design

- **Put the next-state logic in one CASE statement**
  - The state register may also be included here or in a separate process or always block
- **Put the state machine outputs in a separate process or always block**
  - Easier for synthesis tools to optimize logic this way



The Perfect State Machine

- **The perfect state machine has**
  - Inputs: input signals, state jumps
  - Outputs: output states, and control/enable signals to the rest of the design
  - *NO* arithmetic logic, datapaths, or combinatorial functions inside the state machine



State Machine Encoding

- **Use enumerated types to define state vectors (VHDL)**
  - Most synthesis tools have commands to extract and re-encode state machines described in this way
- **Use one-hot encoding for high-performance state machines**
  - Uses more registers, but simplifies next-state logic
  - Experiment to discover how your synthesis tool chooses the default encoding scheme
- **Register state machine outputs for higher performance**

Outline

- Introduction
- Coding Tips
- **Instantiating Resources**
- Summary
- Appendix:
  - Inferring Logic and Flip-flop Resources
  - Inferring Memory
  - Inferring I/Os and Global Resources

Instantiation Tips

- **Use instantiation only when it is necessary to access device features or increase performance or decrease area**
  - Exceptions are noted at the end of this section
- **Limit the location of instantiated components to a few source files, to make it easier to locate these components when porting the code**

Virtex-II Resources

- **Shift register LUT (SRL16 / SRLC16)**
- **F5, F6, F7, and F8 MUX**
- **Carry Logic**
- **MULT_AND**
- **Memories (distributed and block, RAM and ROM)**
- **MULT18x18 / MULT18x18S**
- **SelectI/O™**
- **I/O Registers (Single or Double Data Rate)**
- **Global clock buffers (BUFG, BUFGDLL, BUFGCE, BUFGMUX)**
- **DCM**
- **STARTUP_VIRTEX2**

Synopsys FPGA Compiler II

- **Can be inferred:**
  - Shift register LUT (SRL16 / SRLC16)
  - F5, F6, F7, and F8 MUX
  - Carry Logic
  - MULT_AND
  - Memories (ROM)
  - MULT18x18 / MULT18x18S
  - Global clock buffers (BUFG)
- **Can be inferred using constraints table:**
  - SelectI/O™(single-ended)
  - I/O Registers (Single Data Rate)
  - BUFGDLL*
- **Cannot be inferred:**
  - Memories (RAM)
  - SelectI/O (differential)
  - I/O Registers (Double Data Rate)
  - Global clock buffers (BUFGCE, BUFGMUX)
  - DCM
  - STARTUP_VIRTEX2

Synplicity Synplify Pro 7.1

- **Can be inferred:**
  - Shift register LUT (SRL16 / SRLC16)
  - F5, F6, F7, and F8 MUX
  - Carry Logic
  - MULT_AND
  - Memories (distributed RAM)
  - MULT18x18 / MULT18x18S
  - Global clock buffers (BUFG)

- **Can be inferred using constraints editor or attributes:**
  - Memories (distributed ROM, some block RAM*)
  - SelectI/O™(single-ended)
  - I/O Registers (Single Data Rate)
  - BUFGDLL**

Synplicity Synplify Pro 7.1

- **Cannot be inferred:**
  - Memories (complex block RAM)
  - SelectI/O™ (differential)
  - I/O Registers (Double Data Rate)
  - Global clock buffers (BUFGCE, BUFGMUX)
  - DCM
  - STARTUP_VIRTEX2

Exemplar Leonardo Spectrum 2002b

- **Can be inferred:**
  - Shift register LUT (SRL16 / SRLC16)
  - F5, F6, F7, and F8 MUX
  - Carry Logic
  - MULT_AND
  - Memories (distributed ROM and RAM, some block RAM*)
  - MULT18x18 / MULT18x18S
  - Global clock buffers (BUFG, BUFGDLL**, BUFGCE, BUFGMUX)
- **Can be inferred using constraints editor or attributes:**
  - SelectI/O™ (single-ended)
  - I/O Registers (Single Data Rate)
- **Cannot be inferred**
  - Memories (complex block RAM)
  - SelectI/O (differential)
  - I/O Registers (Double Data Rate)
  - DCM
  - STARTUP_VIRTEX2

XST 5.1i

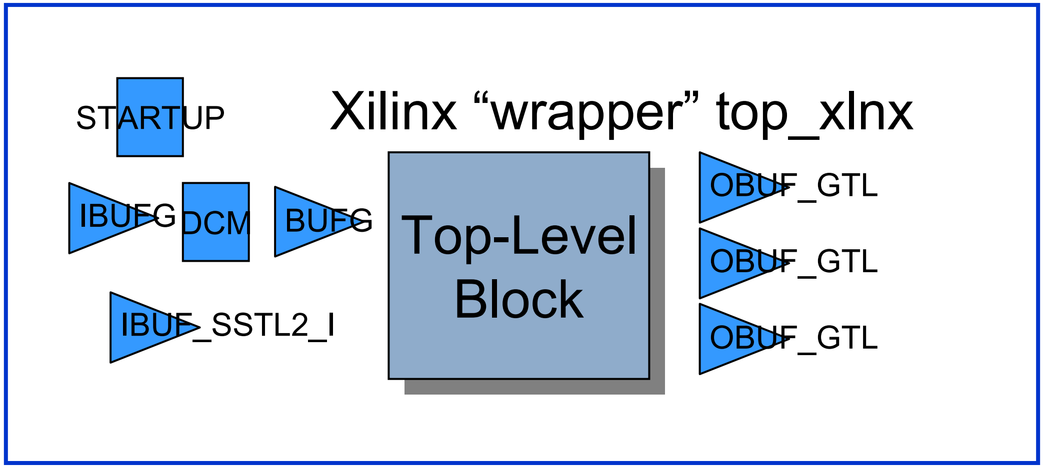
- **Can be inferred:**
  - Shift register LUT (SRL16 / SRLC16)
  - F5, F6, F7, and F8 MUX
  - Carry Logic
  - MULT_AND
  - Memories (distributed ROM and RAM, block RAM*)
  - MULT18x18 / MULT18x18S
  - Global clock buffers (BUFG)
- **Can be inferred using constraints editor or attributes:**
  - SelectI/O™
  - I/O Registers (Single Data Rate)
  - Global clock buffers (BUFGCE, BUFGMUX, BUFGDLL**)
- **Cannot be inferred:**
  - I/O Registers (Double Data Rate)
  - DCM
  - STARTUP_VIRTEX2

Suggested Instantiation

- **Xilinx recommends that you instantiate the following elements:**
  - Memory resources
    - Block RAMs specifically (use CORE Generator™ to build large memories)
  - SelectI/O™ resources
  - Clock management resources
    - DCM (use the Architecture Wizard)
    - IBUFG, BUFG, BUFGMUX, BUFGCE
  - Startup block

Suggested Instantiation

- **Why do we suggest this?**
  - Easier to change (port) to other and newer technologies
  - Fewer synthesis constraints and attributes to pass on
    - Keeping most of the attributes and constraints in the Xilinx UCF file keeps it simple—one file contains critical information



- **Create a separate hierarchical block for instantiating these resources**
  - Above the top-level block, create a Xilinx “wrapper” with Xilinx specific instantiations

Outline

- Introduction
- Coding Tips
- Instantiating Resources
- **Summary**
- Appendix:
  - Inferring Logic and Flip-flop Resources
  - Inferring Memory
  - Inferring I/Os and Global Resources

Review Questions

- What problem occurs with nested CASE and IF/THEN statements?
- Which encoding scheme is preferred for high-performance state machines?
- Which Xilinx resources generally must be instantiated?

Answers

- **What problem occurs with nested CASE and IF/THEN statements?**
  - Nested CASE and IF/THEN statements may generate long delays due to cascaded functions
- **Which encoding scheme is preferred for high-performance state machines?**
  - One-hot
- **Which Xilinx resources generally must be instantiated?**
  - Double Data Rate I/O registers
  - BUFGMUX
  - BUFGCE
  - DCM
  - STARTUP_VIRTEX2

Summary

- Your HDL coding style can affect synthesis results
- Infer functions whenever possible
- Most resources are inferable, either directly or with an attribute
- CASE and IF/THEN statements produce different types of multiplexers
- Avoid nested CASE and IF/THEN statements
- Use one-hot encoding to improve design performance
- When coding a state machine, separate the next-state logic from state machine output equations

Where Can I Learn More?

- ***Synthesis & Simulation Design Guide*: http://support.xilinx.com > Software Manuals**
- **Handbooks: http://support.xilinx.com > Documentation > “Virtex-II Handbook” or “Virtex-II Pro Handbook”**
  - Part 2: *Virtex-II User Guide* > Chapter 2: Design Considerations
    - Using the DCM, memory, etc.
- **Technical Tips: http://support.xilinx.com > Tech Tips**
  - Click Exemplar, Synopsys FPGA Compiler, or Synplicity
- **Answers Database: http://support.xilinx.com > Troubleshoot**
- **Synthesis documentation or online help**

Outline

- Introduction
- Coding Tips
- Instantiating Resources
- Summary
- **Appendix:**
  - **Inferring Logic and Flip-flop Resources**
  - Inferring Memory
  - Inferring I/Os and Global Resources

Shift Register LUT (SRL16)

Synopsys, Synplicity, Exemplar, and XST

- **To infer the SRL, the primary characteristics the code must have are:**
  - No set/reset signal
  - Serial-in, Serial-out
- **SRLs can be initialized on power-up via an INIT attribute in the Xilinx User Constraint File (UCF)**

SRL16E Example

VHDL:

```
process(clk)
begin
    if rising_edge(clk) then
       if ce = ‘1’ then
           sr <= input & sr(0 to 14);
       end if;
    end if;
end process;
output <= sr(15);
```

Verilog:

```
always @ (posedge clk)
begin
        if (ce)
                        sr <= {in, sr[0:14]};
end
assign out <= sr[15];
```

Dynamically Addressable SRL

Synopsys, Synplicity, Exemplar, and XST

- **SRL16/SRL16E, and SRLC16/SRLC16E**
  - SRLC16 has two outputs in Virtex™-II, q15 - final output, and q - dynamically addressable output

SRLC16E Example

VHDL:

```
process(clk)
begin
    if rising_edge(clk) then
      if CE = ‘1’ then
         sr <= input & sr(0 to 14);
      end if;
    end if;
end process;
output <= sr(15);
dynamic_out <= sr(addr);
```

Verilog:

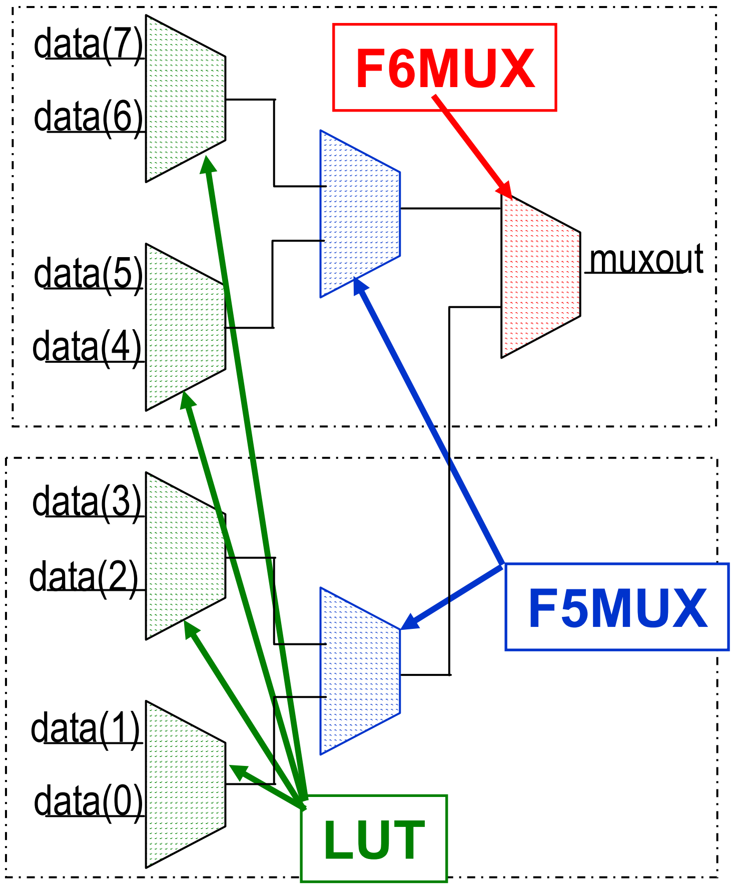
```
always @ (posedge clk)
begin
        if (ce)
                        sr <= {in, sr[0:14]};
end
assign out <= sr[15];
assign dynamic_out <= sr[addr];
```

Virtex-II Multiplexers

Synopsys, Synplicity, Exemplar, and XST

- **F5MUX, F6MUX, F7MUX, F8MUX primitives**
  - Dedicated multiplexers in Virtex™-II CLB
  - Only F5/F6 available in Virtex family
  - 4:1 multiplexer will use one slice
  - 16:1 multiplexer will use 4 slices (1 Virtex-II CLB)
  - 32:1 multiplexer will use 8 slices

- **No attribute needed -- inferred automatically**



F5MUX and F6MUX Example

VHDL:

```
process(sel, data)
begin
    case (sel) is
      when “000” => out <= data(0);
      when “001” => out <= data(1);
      when “010” => out <= data(2);
      when “011” => out <= data(3);
      when “100” => out <= data(4);
      when “101” => out <= data(5);
      when “110” => out <= data(6);
      when “111” => out <= data(7);
      when others => out <= ‘0’;
    end case;
end process;
```

Verilog:

```
always @ (sel or data)
case(sel)
        3'b000: muxout = data[0];
        3'b001: muxout = data[1];
        3'b010: muxout = data[2];
        3'b011: muxout = data[3];
        3'b100: muxout = data[4];
        3'b101: muxout = data[5];
        3'b110: muxout = data[6];
        3'b111: muxout = data[7];
        default : muxout = 0;
endcase
```

Flip-flop Set/Reset Conditions

- **When using asynchronous set and asynchronous reset, reset has priority**
- **When using synchronous set and synchronous reset, reset has priority**
- **When using any combination of asynchronous set/reset with synchronous set/reset:**
  - Asynchronous set/reset has priority (furthermore, reset has *highest* priority)
  - In this mode, the synchronous set and/or reset is implemented in the LUT
  - The priority of the synchronous set versus synchronous reset is defined by how the HDL is written

Flip-Flop Example

VHDL:

```
process(clk, reset, set)
begin
        if (reset = ‘1’) then q <= ‘0’;
        elsif (set = ‘1’) then q <= ‘1’;
        elsif rising_edge(clk) then
            if (sync_set = ‘1’) then
                        q <= ‘1’;
            elsif (sync_reset = ‘1’) then
                        q <= ‘0’;
            elsif (ce = ‘1’) then
                        q <= d;
            end if;
        end if;
end process;
```

Verilog:

```
always @ (posedge clk or posedge
reset or posedge set)
        if (reset)
                        q = 0;
        else if (set)
                        q = 1;
        else if (sync_set)
                        q = 1;
        else if (sync_reset)
                        q = 0;
        else if (ce)
                        q = d;
end
```

Carry Logic

Synopsys, Synplicity, Exemplar, and XST

- **Synthesis maps directly to the dedicated carry logic**
- **Access carry logic through adders, subtractors, counters, comparators (>15 bits) and other arithmetic operations**
  - Adders / subtractors (SUM <= A + B)
  - Comparators (if A < B then)
  - Counters (COUNT <= COUNT + 1)
- **Note: Carry logic will not be inferred if arithmetic components are built with gates**
  - For example: XOR gates for addition and an AND gate for carry logic will not infer carry logic

Carry Logic Examples

VHDL:

```
count <= count + 1 when
        (addsub = ‘1’) else count - 1;

if (a >= b) then
        a_greater_b <= ‘1’;

product <= constant * multiplicand;
```

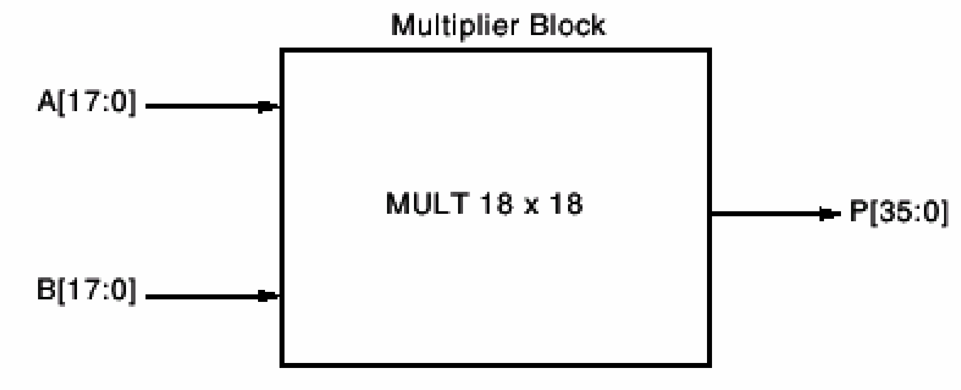
Verilog:

```
assign count = addsub ? count + 1:
count - 1;

if (a >= b)
        a_greater_b = 1;

assign product = constant *
multiplicand;
```

MULT18x18

Synopsys, Synplicity, Exemplar, and XST

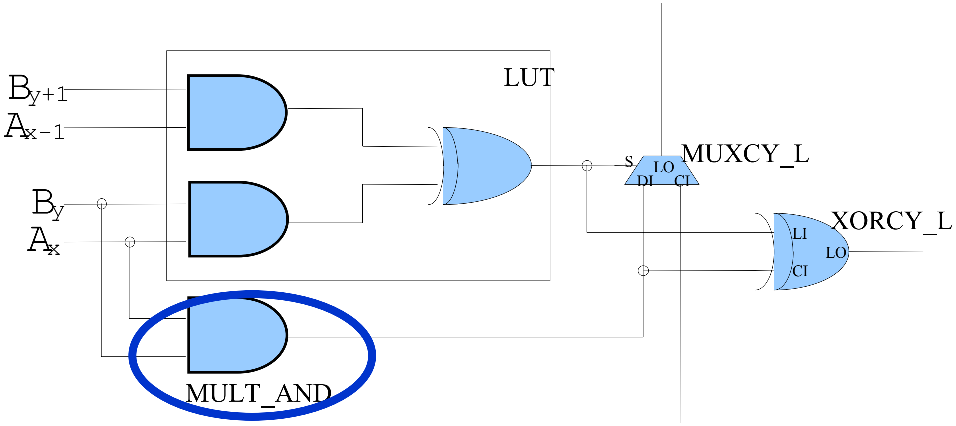
- **Synplicity and Exemplar infer MULT18x18 by default**
  - Synplify, to use MULT_18x18 set:
    - syn_multstyle = block_mult (default)
      - Possible values are “block_mult” and “logic”
  - Exemplar, to use MULT_18x18 set:
    - virtex2_multipliers = true (default)



CLB MULT_AND

Synopsys, Synplicity, Exemplar, and XST

- **Synplicity set: *syn_multstyle* = *logic***
- **Exemplar set: *virtex2_multipliers* = *false***
- **Synopsys will use MULT_AND by default**



Multiplier Example

VHDL:

```
library ieee;
use ieee.std_logic_signed.all;
use ieee.std_logic_unsigned.all;
…
process (clk, reset)
begin
        if (reset = ‘1’) then
            product <= (others => ‘0’);
        elsif rising_edge(clk) then
            product <= a * b;
        end if;
end process;
```

Verilog:

```
always @ ( posedge clk or posedge
reset)
begin
        if (reset)
                        product <= 0;
        else
                        product <= a * b;
end
```

Outline

- Introduction
- Coding Tips
- Instantiating Resources
- Summary
- **Appendix:**
  - Inferring Logic and Flip-flop Resources
  - **Inferring Memory**
  - Inferring I/Os and Global Resources

Block SelectRAM

Synplicity, Exemplar, and XST

- **Synplicity: Set the syn_ramstyle attribute to “block_ram”**
  - Place the attribute on the output signal driven by the inferred RAM
  - Requires synchronous write
  - Requires registered read address
  - Dual-port RAM inferred if read/write address index is different
- **Exemplar: Automatically inferred under these conditions:**
  - Synchronous write
  - Registered read address
- **XST: Based on the size and characteristics of the code, XST can automatically select the best style**
  - Available settings: Auto, Block, Distributed

Block RAM Inference Notes

Synplicity, Exemplar, and XST

- **Synthesis tools cannot infer:**
  - Dual-Port block RAMs with configurable aspect ratios
    - Ports with different widths
  - Block RAMs with enable or reset functionality
    - Always enabled
    - Output register cannot be reset
  - Dual-Port block RAMs with read and write capability on both ports
    - Block RAMs with read capability on one port and write on the other port can be inferred
  - Dual-Port functionality with different clocks on each port
- **These limitations on inferring block RAMs can be overcome by creating the RAM with the CORE Generator™ or instantiating primitives**

Block RAM Example

VHDL:

```
signal mem: mem_array;
attribute syn_ramstyle of mem: signal
is “block_ram”;
…
process (clk)
begin
        if rising_edge(clk) then
            addr_reg <= addr;
            if (we = ‘1’) then
                        mem(addr) <= din;
            end if;
        end if;
end process;
dout <= mem(addr);
```

Verilog:

```
reg [31:0] mem[511:0] /*synthesis
syn_ramstyle = “block_ram”*/;

always @ ( posedge clk)
begin
        addr_reg <= addr;
        if (we)
                        mem[addr] <= din;
end

assign dout = mem[addr_reg];
```

Distributed SelectRAM

Synplicity, Exemplar, and XST

- **Each LUT can implement a 16x1-bit synchronous RAM**
- **Automatic inference when code is written with two requirements:**
  - Write must be synchronous
  - Read must be asynchronous
    - However, if the read address is registered, the SelectRAM™ can be inferred and will be driven by a register
  - Synplicity: Automatically used -- turn off by setting attribute:
    - syn_ramstyle = registers or block_ram
  - Exemplar: Automatically used if RAM is less than eight bits wide
    - Or set *block_ram* = *false* on RAM output signal
  - XST: Specify block or distributed RAM, or let XST automatically select the best implementation style

Distributed RAM Example

VHDL:

```
signal mem: mem_array;
…
process (clk)
begin
        if rising_edge(clk) then
            if (we = ‘1’) then
                        mem(addr) <= din;
            end if;
        end if;
end process;
dout <= mem(addr);
```

Verilog:

```
reg [7:0] mem[31:0];

always @ ( posedge clk)
begin
        if (we)
                        mem[addr] <= din;
end

assign dout = mem[addr];
```

ROM

Synopsys, Synplicity, Exemplar, and XST

- **Synplicity: infer ROM primitives with an attribute**
  - Set syn_romstyle = select_rom
  - Otherwise, Synplify will infer a LUT primitive with equations
    - Same implementation, except it is not a ROM primitive
- **Exemplar, Synopsys, and XST will automatically map to ROM primitives**

Distributed ROM Example

VHDL:

```
type rom_type is array(7 downto 0) of
std_logic_vector(1 downto 0);
constant rom_table: rom_type :=
(“10”, “00”, “11”, “01”, “11”, “10”,
“01”, “00”);
attribute syn_romstyle: string;
attribute syn_romstyle of rom_table:
signal is “select_rom”;
…
rom_dout <= rom_table(addr);
```

Verilog:

```
reg [1:0] rom_dout /*synthesis
syn_romstyle = “select_rom”*/;

always @ ( addr)
case (addr)
        3’b000: rom_dout <= 2’b00;
        3’b001: rom_dout <= 2’b01;
        3’b010: rom_dout <= 2’b10;
        3’b011: rom_dout <= 2’b11;
        3’b100: rom_dout <= 2’b01;
        3’b101: rom_dout <= 2’b11;
        3’b110: rom_dout <= 2’b00;
        3’b111: rom_dout <= 2’b10;
endcase
```

Outline

- Introduction
- Coding Tips
- Instantiating Resources
- Summary
- **Appendix:**
  - Inferring Logic and Flip-flop Resources
  - Inferring Memory
  - **Inferring I/Os and Global Resources**

SelectI/O

Synopsys, Synplicity, and Exemplar

- **Instantiate in HDL code (required for differential I/O)**
  - For a complete list of buffers, see the following elements in the “Libraries Guide”:
    - IBUF_selectIO, IBUFDS
    - IBUFG_selectIO, IBUFGDS
    - IOBUF_selectIO
    - OBUF_selectIO, OBUFT_selectIO, OBUFDS, OBUFTDS
- **Use attribute (Synplicity, Exemplar)**
- **Specify in the UCF file**
- **Use Xilinx Constraints Editor**
  - In the Ports tab, check the I/O Configuration Options box

SelectI/O

- **Synopsys: Ports tab of FPGA Compiler II constraints editor**
- **Synplicity: Use *xc_padtype* attribute**
- **Exemplar: Use *buffer_sig* attribute**
- **XST: Instantiate or use Xilinx Constraints Editor**

SelectI/O Example

VHDL:

```
component IBUF_HSTL_III
    port (I: in std_logic;
          O: out std_logic);
end component;
...
ibuf_data_in_inst: IBUF_HSTL_III
    port map (I => data_in, O =>
    data_in_i);
```

Verilog:

```
/* For primitive instantiations in
Verilog you must use UPPERCASE
for the primitive name and port
names */

IBUF_HSTL_III ibuf_data_in_inst
       (.I(data_in), .O(data_in_i));
```

IOB Registers

Synopsys, Synplicity, Exemplar, and XST

- **For Single Data Rate I/O registers:**
  - Set Map Process Properties > Pack I/O Registers/Latches into IOBs
  - Use the IOB = TRUE attribute in the UCF file
    - Use on instantiated FFs or inferred FFs with known instance name
    - Example: INST <FF_instance_name> IOB = TRUE;
  - Synopsys: Ports tab in FPGA Compiler II constraints editor
  - Synplicity: Automatically packs registers in the IOB, based on timing
    - To override default behavior, use the syn_useioff attribute
  - Exemplar: virtex_map_iob_registers = TRUE
  - XST: Automatically packs registers in the IOB, based on timing
    - To override default behavior under Synthesize > Properties > Xilinx Specific Options tab > Pack I/O Registers into IOBs > Auto, Yes, or No

IOB Registers

Synopsys, Synplicity, Exemplar, and XST

- **Double Data Rate registers must be instantiated**
  - See the following elements in the “Libraries Guide”:
    - IFDDRCPE, IFDDRRSE (for input flip-flops)
    - OFDDRCPE, OFDDRRSE (for output or 3-state enable flip-flops)
    - OFDDRTCPE, OFDDRTRSE (for output flip-flops)

IOB Registers

- **Limitations:**
  - All flip-flops that are packed into the same IOB must share the same clock and reset signal
    - They can have independent clock enables
  - Output and 3-state enable registers must have a fanout of one
    - Synopsys and Synplicity will automatically replicate 3-state enable registers to enable packing into IOBs
    - Exemplar can replicate 3-state enable registers for packing into IOBs by setting virtex_map_iob_registers = TRUE
  - Output 3-state enables must be active-low
    - There is logic in the IOB to invert 3-state enable signal **before** the register
  - DDR registers must use clk and not clk with 50 percent duty-cycle or DLL outputs clk0 and clk180

I/O Register Example

VHDL:

```
process(clk, reset)
begin
    if (reset = ‘1’) then
      data_in_i <= ‘0’;
      data_out <= ‘0’;
      out_en <= ‘1’;
    elsif rising_edge(clk) then
      data_in_i <= data_in;
      out_en <= out_en_i;
      if (out_en = ‘0’) then
         data_out <= data_out_i;
      end if;
    end if;
end process;
```

Verilog:

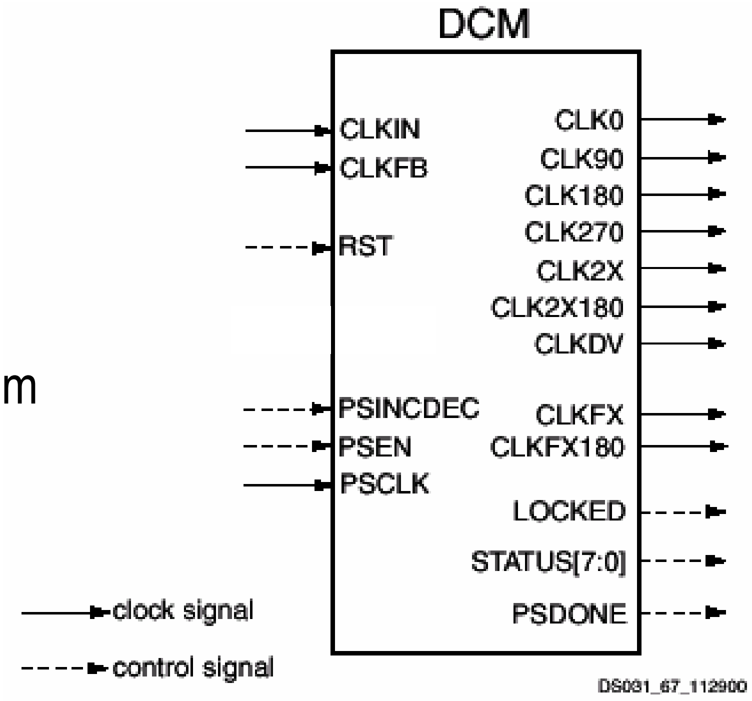
```
always @ (posedge clk or posedge reset)
      if (reset)
      begin
         data_in_i <= 0;
         data_out <= 0;
         out_en <= 1;
      end
      else
      begin
         data_in_i <= data_in;
         out_en <= out_en_i;
         if (~out_en)
                  data_out <= data_out_i;
      end
```

Global Buffers

- **BUFG**
  - All synthesis tools will infer on input signals that drive the clock pin of any synchronous element
- **BUFGDLL**
  - Synopsys: Specify in the Ports tab of FPGA Compiler II constraints editor
  - Synplicity: Can be inferred through synthesis by setting attribute xc_clockbuftype = BUFGDLL
  - Exemplar: Can be inferred through synthesis by setting attribute PAD = BUFGDLL or BUFGDLLHF
  - XST: Must instantiate
- **BUFGCE**
  - Exemplar: Can be inferred by setting virtex2_clock_mux = TRUE

DCM

- **Digital Clock Manager**
  - Clock de-skew
  - Frequency synthesis
  - Phase shifting
- **Must be instantiated**
  - Port names are as shown in diagram



STARTUP_VIRTEX2

- **Provides three functions**
  - Global set/reset (GSR)
  - Global Three-State for output pins (GTS)
  - User-defined configuration clock to synchronize configuration startup sequence
- **Must be instantiated**
  - Port names are GSR, GTS, and CLK
- **Note: Using GSR is not recommended for Virtex™-II designs**
  - Normal routing resources are faster and plentiful